Direct numerical simulation of turbulence on a connection machine CM-5

Verstappen, R.W.C.P.; Veldman, A.E.P.

Published in:
Applied numerical mathematics

Publication date:
1995

ELSEVIER

Applied Numerical Mathematics 19 (1995) 147-158

APPLIED NUMERICAL MATHEMATICS

Direct numerical simulation of turbulence on a Connection Machine CM-5

R.W.C.P. Verstappen *, A.E.P. Veldman

Department of Mathematics, University of Groningen, P.O. Box 800, 9700 AV Groningen, Netherlands

Received 7 November 1994

Abstract

In this paper we report on our first experiences with direct numerical simulation of turbulent flow on a 16-node Connection Machine CM-5. The CM-5 has been programmed at a global level using data parallel Fortran. A two-dimensional direct simulation, where the pressure is solved using a Conjugate Gradient method without preconditioning, runs at 23% of the peak. Due to higher communication costs, 3D simulations run at 13% of the peak. A diagonalwise re-ordered Incomplete Choleski Conjugate Gradient method cannot compete with a standard CG-method on the CM-5.

1. Introduction

Computer simulation has become a major tool to study turbulent flows. In many technological applications, unfortunately, direct numerical simulation (DNS) of turbulent flows—i.e., computing numerical solutions of the unsteady 3D Navier–Stokes equations that resolve the evolution of all dynamically significant scales of motion—more than exhausts the presently largest available computing resources by requiring machines in the exa(10^{18})flops range with exabytes of memory. Thus, for turbulent engineering flows, acceptable computational effort can only be obtained by modeling the turbulent motion of the small scales in the flow.

Turbulence modeling forms the Achilles' heel of applied computational fluid dynamics: with existing turbulence models the simulation accuracy required by industry cannot always be reached. It is generally expected that DNS will play a key role in obtaining reasonable accurate turbulence models for applied computational fluid dynamics. For an overview of the impact of DNS on turbulence modeling and research, see for instance [1].

The enormous appetite for floating-point operations and bytes limit DNS to low Reynolds numbers. Flows are only weakly turbulent at these Reynolds numbers. Before the turn of the century, massively

* Corresponding author. E-mail: R.W.C.P.Verstappen@math.rug.nl.

SSDI 0168-9274(95)00022-4

Table 1

Reynolds number	Grid points	Memory	CPU time	
			300 hours	0.3 hour
10^4	10^7–10^8	1–10 Gb	1–10 Gflops	1–10 Tflops
10^7–10^8	10^{15}–10^{17}	10^5–10^7 Tb	10^5–10^7 Tflops	–

parallel machines will offer the Teraflop performance. Nowadays, already, the high end of the CM-5 line has a peak of 0.13 Teraflop/s. This figure is certainly impressive, but what can we get out of a CM-5? To answer this question, we have investigated how a 16-node CM-5 performs on an existing DNS-code, that has been developed for use on vector computers (in particularly CRAY YMP and NEC SX-3). Here, we have restricted ourselves to data parallel Fortran. Related questions that we will consider read: which parts of our approach have to be altered to obtain a faster implementation on the CM-5; can a 16-node CM-5 be viewed as a production machine, i.e., is the working speed of a 16-node CM-5 (peak 2 Gflop/s) comparable to that of one vector processor of NEC's SX-3 (peak 2.7 Gflop/s), e.g.

In the following section we will describe the main characteristics of the Connection Machine CM-5 shortly. In Section 3, the computational requirements for DNS of turbulent flows are outlined and an example—DNS of turbulent flow in a driven cavity—is given. The computational procedure is outlined in Section 4. Its parallelization is discussed in Section 5. Conclusions will be drawn in Section 6.

2. The Connection Machine CM-5

The Connection Machine CM-5 as installed at the University of Groningen is a 16 processor node system with a peak performance of approximately 2 Gflop/s. Each processing node is a 128 Mflop/s computational unit composed of a SPARC micro-processor, 4 vector units, 32 Mbytes of memory and a network interface. The structure of data network is a so-called fat tree. The data network guarantees 10 Mbyte/s to each processing node, no matter where in the system the data is being sent.

The CM-5 can be programmed both on a global level (using data parallel Fortran or data parallel C) and on a local level (in a message-passing programming style). We restrict ourselves to the global level of programming and use CM-Fortran, which is practically identical to the language Fortran 90. Large parts of the CM-Fortran code have been generated from an existing Fortran 77 program by using CMax, which is a tool that automatically converts Fortran 77 programs to CM-Fortran.

3. Direct numerical simulation: future and present

As already stated in the introduction, high performance computing is a prerequisite for direct numerical simulation of turbulence. Table 1 gives an overview of the requirements, in terms of processing power and memory size, for DNS of flows in an early stage of turbulence (Reynolds' number $Re = 10^4$), and for DNS of fully developed turbulent flows (see also [1] or [2]).

Nine orders of magnitude have to be bridged to perform a DNS of a fully developed turbulent flow. Assuming that both computer hardware and computational algorithms will continue to progress

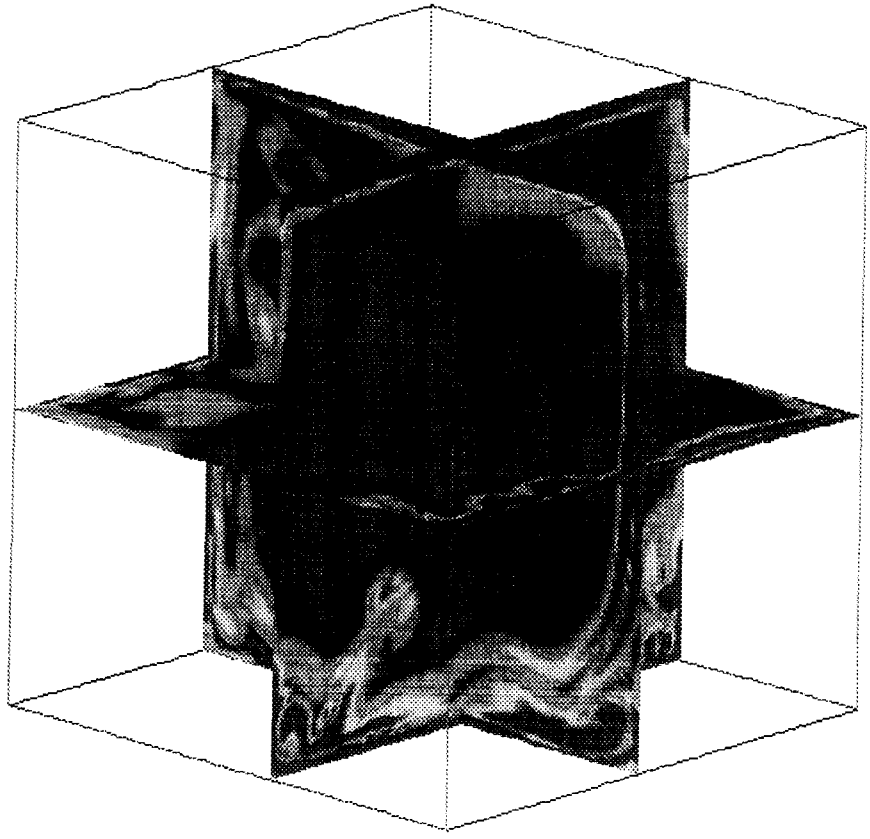

Fig. 1. Snapshot of the vorticity in a 3D driven cavity at $Re = 10,000$. The orientation of the cavity is such that the upperplane is driven from the left/back to the right/front.

at the rate that they have got ahead during the past 30 years—both have become 30 times faster per decade—it will take (at least) three decades to bridge the lacking nine orders of magnitude. For this estimate to come true, computers need to become 30^3 times as fast and need to have 30^3 times more memory within the next 30 years; the numerical algorithms of 2024 need to be 30^3 times faster than the present ones, need to run efficiently at the fastest 2024-machines, and need to use 30^3 times less memory than todays algorithms do require. The latter is not often mentioned. Yet, today already, the available number of Mbytes does restrict the size of direct simulations. One way to overcome this, is by using domain-swapping techniques.

Presently, direct numerical simulations of turbulent flows are restricted to (relatively) low Reynolds numbers. About $Re = 10,000$ is the highest attainable Reynolds number for a DNS. As an example, Fig. 1 shows an instantaneous vorticity field in a 3D cubical lid-driven cavity at $Re = 10,000$, as obtained by DNS. The results of this simulation agree well with the available experimental data. That is, the DNS reproduces the experimentally observed Taylor–Görtler-like vortices, and numerically and experimentally obtained mean velocities, root-mean-square velocities and power spectra do agree well. The DNS provides much more detailed information about this turbulent flow than the experiments do, and can be utilized to validate turbulence models for recirculating flow. For more details see [3].

4. The computational procedure

To make this paper self contained, the method that has been used to simulate transition and turbulence in a driven cavity is described concisely in this section. For a more detailed discussion of the computational procedure the reader is referred to [3]. The incompressible Navier–Stokes equations are discretized using a finite-volume method, where the velocities and pressures are defined on a staggered grid (cf. [4]). The pressure term and the incompressibility constraint are integrated implicitly in time; the convective and diffusive terms are treated explicitly. The computation of one time step can be divided into three substeps.

First, an intermediate velocity $\breve{u}$ is computed by integrating the convective and diffusive terms of the momentum equations over one time step Δt

$$\tilde{u} = \frac{3u^n}{2} - \frac{u^{n-1}}{2}, \tag{1}$$

$$\breve{u} = u^n + \Delta t \left(- (\tilde{u} \cdot \nabla_h)\, \tilde{u} + \frac{1}{Re} \Delta_h u^n \right). \tag{2}$$

Here, u^n denotes the (given) discrete velocity at time $t = n\,\mathrm{d}t$. The spatial discretizations of the convective and diffusive term are represented by $(\tilde{u}\cdot\nabla_h)\tilde{u}$ and $\Delta_h u^n/Re$ respectively. The discretization of the convective term depends on both u^n and u^{n-1} since a second-order Adams–Bashforth method is used to integrate the convective term in time.

Next, the pressure p^{n+1} at time level $t = (n+1)\Delta t$ is computed from the Poisson equation

$$-\mathrm{div}_h \nabla_h p^{n+1} = -\mathrm{div}_h \breve{u}/\Delta t. \tag{3}$$

And, finally, the divergence-free velocity u^{n+1} is obtained by adding the pressure term to the intermediate velocity $\breve{u}$,

$$u^{n+1} = \breve{u} - \Delta t \nabla_h p^{n+1}. \tag{4}$$

Eqs. (1)–(4) hold in the interior of the spatial domain. At the boundaries, Dirichlet conditions for the velocity are valid.

5. Parallelization of the computational procedure

Solving the Poisson equation (3) takes most of the computing time (no matter how it is done). Hence, this part of the computational procedure should be implemented as efficient as possible. In Section 5.2, we will consider Conjugate Gradient methods for solving the discrete Poisson equation for the pressure. Before, in Section 5.1, we will consider the parallelization of the substeps (1), (2) and (4) of the time-marching procedure.

5.1. Time integration of the convection–diffusion equation

The computation of the substeps (1), (2) and (4) of the explicit time-marching procedure can be done in parallel by letting each processor treat its own subdomain. In the sequel, we will focus on the

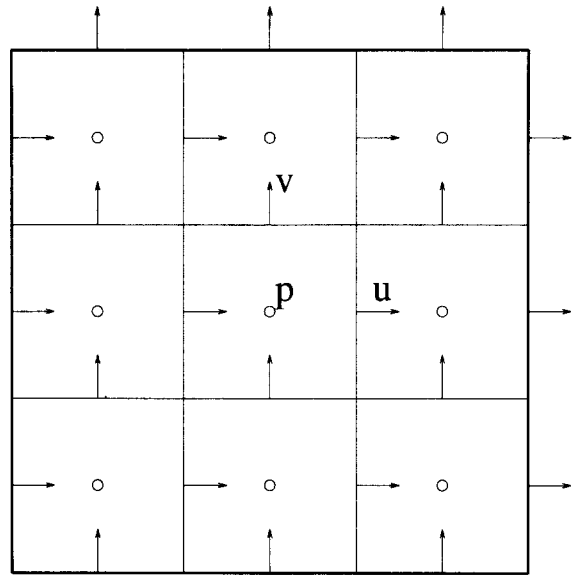

Fig. 2. The location of the discretized velocity (u,v) and pressure p on the staggered grid (in two spatial dimensions).

data parallel execution of (2); the parallelization of (4) goes along the same lines; the parallelization of (1) is trivial. We kick off by discussing various ways to compute (2) data parallel. Next, we will analyze the fastest way by comparing the times for communication and computation.

We have started by simply converting our existing Fortran 77 code into a CM-Fortran code, using the automatic Fortran 77 to CM-Fortran converter called CMax (a CM-5 software tool). The thus obtained data parallel Fortran code performed rather disappointingly: its megaflop rate lies within the range of a $10,000 workstation. The reason for this is that the staggered location of the components of the velocity and the pressure is not recognized by the software. All do-loops are simply replaced by FORALL statements. This CM-Fortran construct is much slower than other constructs that can do the same job, a WHERE statement, or a MERGE statement, e.g. Before discussing these faster constructs, we will consider the data structure in detail.

The flow domain is divided into finite volumes. The discrete pressure is defined at the centre of each volume; the discrete velocity components are defined at the cell faces, namely such that the velocity component perpendicular to a cell face is defined at the middle of that cell face. The staggering of the grid is sketched in Fig. 2.

The computation of velocity components at the boundaries differs from those at internal grid points. The velocities at the boundaries are prescribed by time-independent Dirichlet conditions, i.e., need not to be updated during the time integration. Velocities at internal grid points are to be updated according to (2). This can be realized by a WHERE statement of the following form

$$\texttt{WHERE (''not at the boundary'')} \qquad \breve{u} = u^n + \Delta t \left(- (\tilde{u} \cdot \nabla_h)\, \tilde{u} + \frac{1}{Re} \Delta_h u^n \right).$$

The condition "not at the boundary" differs for the three components of the velocity, due to the staggering of the grid. Thus, three masks are to be constructed: for each component of the velocity one. These masks are independent of time, i.e., need to be computed only once, provided that there

is enough space available to store them during the whole time integration. Instead of using a WHERE statement, the update of the components of the velocity can be computed using array sections, or alternatively, the update can be done unconditionally, followed by a reparation of the violated boundary conditions. All these three solutions are significantly faster than CMax's solution, i.e., than a FORALL statement.

We have compared all above mentioned solutions and found that the unconditional update followed by a reparation of the conditions at the boundaries is the fastest and uses the fewest memory. It is approximately twice as fast as the solution using a WHERE statement, and it is more than an order of magnitude faster than the FORALL solution.

We have implemented the fastest solution. Hereto, another consequence of the staggering of the grid has to be considered. Namely that the three arrays—say u, v and w—containing the three components of the discrete velocity and the array p of discrete pressures are not conformable, i.e., their dimensions differ. Indeed, take nx volumes in the first spatial direction (the velocity component in this direction is denoted by u), ny volumes in second direction (velocity component v) and nz volumes in the third direction (w). Then, the dimensions of the arrays u, v, w and p become

```
u(0:nx,1:ny,1:nz),    v(1:nx,0:ny,1:nz),

w(1:nx,1:ny,0:nz),    p(1:nx,1:ny,1:nz).
```

Adding two nonconformable arrays, for instance u and v, makes no sense in CM-Fortran (nor in Fortran 90). This also holds for other operations. Therefore, all four arrays u, v, w and p are redefined such that they become conformable. That is, all dimensions are taken equal to nx*ny*nz. The "missing" elements, which correspond to prescribed velocities at the boundaries, are stored separately. Then, all updates can be performed unconditionally, i.e., for i=1,...,nx, j=1,...,ny and k=1,...,nz, and the thus violated boundary conditions can be repaired afterwards. It may be noted that this solution is rather laborious for the programmer, since it involves a change of the data structure.

To estimate the ratio between the time needed for communications and time taken by the computations, we will count the number of shifts and floating-point operations needed to integrate the convective–diffusive part of the Navier–Stokes equations over one time step.

Shifting is an intrinsic operation of CM-Fortran (and also of Fortran 90). In fact there are two types of shift, called CSHIFT and EOSHIFT. The "C" in CSHIFT stands for circular; "EO" means end-off. The EOSHIFT allows one to incorporate Dirichlet boundary conditions; the CSHIFT assumes periodicity in the direction of the shift. For instance, let a(1:nx,1:ny) and b(1:nx,1:ny) be two-dimensional, conformable arrays, then the statement

```
b = CSHIFT(a, DIM=2, SHIFT=1)
```

causes the elements of b to become equal to

```
b(i,j) = a(i,j+1)
```

for i=1,...,nx and j=1,...,ny-1 and

```
b(i,ny) = a(i,1)
```

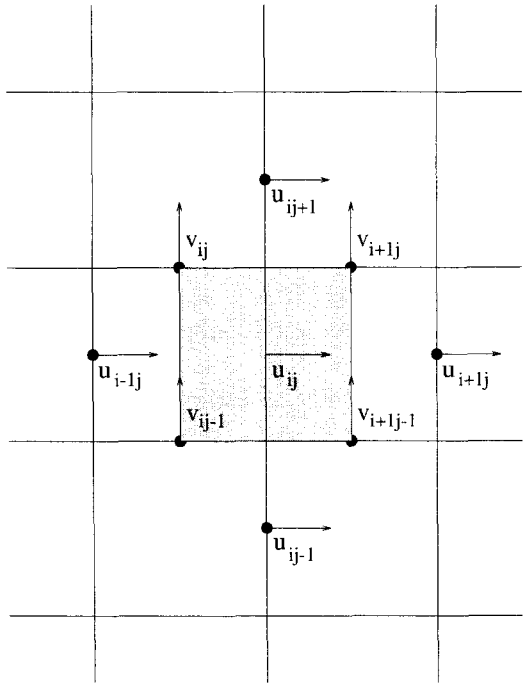

Fig. 3. Array elements of u and v needed to update u_{ij}.

for i=1,...,nx. The integration of the discretized convective–diffusive terms of the Navier–Stokes equations requires nearest-neighbor shifts only. Fig. 3 shows the elements that are involved in the update of an element u_{ij}. A similar figure can be drawn in three spatial dimensions, and for the other components of the velocity vector. The second-order central discretization of the diffusive part of the two-dimensional Navier–Stokes equation results into the well-known five-point molecule. Its evaluation requires four shifts (by plus and minus one in the first direction, and by plus and minus one in the second direction). On a stretched grid, all five elements of the stencil have to be multiplied by

Table 2

2D	Convection	Diffusion	Conv. + diff.
shifts	7	4	11
flops	28	9	38
ratio	0.25	0.44	**0.29**

Table 3

3D	Convection	Diffusion	Conv. + diff.
shifts	12	6	18
flops	44	13	58
ratio	0.27	0.46	**0.31**

different constants and have to be summed together. This costs nine floating-point operations. Thus, in two spatial dimensions, the ratio between shifts and floating-point operations for the diffusive part is approximately equal to 0.44. This ratio can also be determined for the evaluation of the convective term of the Navier–Stokes equation. The results are summarized in Tables 2 and 3 for two and three spatial dimensions, respectively. N.b. In these two tables, the number of shifts and flops is counted per equation. Note that there are two momentum equations, for both components of the velocity one, in 2D and three in 3D.

As can be seen from the two tables, the ratio between (nearest-neighbor) shifts and floating-point operations, i.e., the ratio between communication and computation, for one evaluation of a second-order finite-volume discretization of a convection–diffusion equation is approximately 0.3 in both two and three spatial dimensions.

From this ratio, the communication costs can be estimated. To obtain an estimation of the communication costs, we will count the number of data elements that have to be moved from a local memory to another for one shift first. Here, we consider the operation `CSHIFT(u,DIM=1,SHIFT=-1)` on a target machine with p local memories. The array u has N entries. It can either be a two- or a three-dimensional array. In two dimensions it is defined as $u(1:N_2,1:N_2)$, where $N_2^2 = N$; in 3D we take $u(1:N_3,1:N_3,1:N_3)$, with $N_3^3 = N$. To ease the counting, we assume that the 2D array u can be divided into p subarrays of size $(N_2/p^{1/2})^2$, and that all elements of one subarray are stored in one local memory. Likewise, we assume that the 3D array u consists of p subarrays of size $(N_3/p^{1/3})^3$, and that the elements of one subarray are located in one local memory. Then, the absolute number of data motions is equal to $(N_2/p^{1/2})*p$ in 2D and $(N_3/p^{1/3})^2*p$ in 3D. The relative number of data motions are $(p/N)^{1/2}$ and $(p/N)^{1/3}$ respectively.

Each node of a 16-node CM-5 has 4 vector units, and each vector unit has its own local memory. Thus, in total, there are 64 local memories. The 16 nodes can communicate at a speed of 10 Mb/s. Two local memories within one node communicate at 20 Mb/s. Hence, the average speed of communication is $0.25*10+0.75*20 = 17.5$ Mb/s. We take 10^6 data elements of 8 bytes each. Then, the total number of bytes to be moved for the evaluation of one convection–diffusion equation in two spatial dimensions can be estimated by $8 \cdot 10^3 * 11 * 8 = 704000$. This data motion takes approximately 0.04 seconds (at a speed of 17.5 Mb/s). We have measured the actual time that the shifts take, and found that the actual time equals the estimation: both come to 0.04 seconds per equation.

As remarked before, the ratio between shifts and flops is approximately equal to 0.3. Now, let us assume that the flops are for free, i.e., that they can be overlapped with the communications. Then, $38 \cdot 10^6$ floating-point operations would take 0.04 seconds, and the time integration of the 2D convection–diffusion equation would run at 950 Mflop/s (46% of peak). This thought experiment shows that the communication slows down the performance. It goes without saying that the actual Mflop rate has been measured: the time integration of the convection–diffusion equation on 1000^2 grid runs at approximately 15% of the peak. Thus, the ratio between the communication time and computation time equals 1 to 2.

In three spatial dimensions using a 100^3 grid, we have measured a time of 0.6 seconds for the shifts required to evaluate three convection–diffusion equations. This limits the speed to 300 Mflop/s (15% of peak), where the maximum can only be reached if the flops are fully overlapped with the communications. The latter is not the case: the actual speed is approximately 7% of the peak, i.e., the communication time equals the time of the computations.

We conclude this section, by summarizing the main results in Table 4.

Table 4

	Speed		Computation : communication
	Mflops	% peak	
2D	300	15%	2:1
3D	150	7%	1:1

Table 5

	CRAY YMP (one processor) 333 Mflop/s peak		NEC SX-3 (one processor) 2750 Mflop/s peak	
	Mflop/s	% peak	Mflop/s	% peak
CG	260	78%	1100	40%
ICCG	175	54%	700	25%

5.2. *How to solve the Poisson equation in parallel?*

Solving the pressure from the Poisson equation (3) is by far the most costly part of the time-stepping procedure. In other words, the performance of an incompressible Navier–Stokes solver is dominated by the performance of the solution technique for the Poisson equation. Consequently, our main task reads: solve the Poisson equation (3) as fast as possible.

In [3], an Incomplete Choleski Conjugate Gradient method has been used to solve 100^3 unknown discrete pressures per time step. The initial guess for the ICCG iteration is obtained by extrapolating the pressure from three previous time levels. The ICCG method requires 390 floating-point operations per unknown discrete pressure.

The preconditioner in [3] is constructed from an incomplete Choleski decomposition without fill-in. This decomposition is modified according to Gustafson [5]. The preconditioner is time independent and is computed only once, namely before the time-stepping starts. Consequently, the time needed to construct the preconditioner is insignificant. By using Eisenstat's implementation, the preconditioned system can be solved iteratively for practically the same cost as the unpreconditioned system [6].

In two spatial dimensions the preconditioned Poisson system requires 22 floating-point operations per grid point and iteration; the unpreconditioned CG-iteration requires 19. Thus, in terms of floating-point operations per iteration the preconditioner comes almost for free. Yet, in terms of CPU time per iteration the preconditioned CG iteration is more expensive than the unpreconditioned CG iteration. This is due to the fact that the floating-point operations of the ICCG are done at a lower speed. Table 5 illustrates this for two vector computers, that can be viewed as production machines for direct numerical simulations.

The use of the preconditioner reduces the number of iterations needed to converge. The net gain of the use of the preconditioner is about a factor of three. Here, we have counted for the reduction of the number of iterations, the reduction of computational speed, and the slight increase of floating-point operations per iteration.

So far for solving the Poisson equation on vector computers. We now turn to the CM-5. The computation of the right-hand side of the Poisson equation (3) can be done in parallel using the same constructs as used for (1), (2) and (4). The parallelization of the Poisson solver itself is more difficult. Here, we restrict ourselves to two spatial dimensions and consider the discretization given by the standard five-point molecule on a uniform grid.

To start we consider the unpreconditioned Conjugate Gradient method. The data parallel code for this method reads:

```
p = initial guess, r = initial residual,  s = 0, beta = 0
rho = SUM(r*r)
WHILE ( rho .GT. tolerance ) DO
s = r + beta*s
q =  - CSHIFT(s, DIM=1, SHIFT=-1) - CSHIFT(s, DIM=1, SHIFT=1)
     - CSHIFT(s, DIM=2, SHIFT=-1) - CSHIFT(s, DIM=2, SHIFT=1) + 4*s
alpha = rho/SUM(s*q)
p = p + alpha*s
r = r - alpha*q
rhon = SUM(r*r)
beta = rhon/rho
rho = rhon
END WHILE
```

Here, p, r, s and q are arrays (all have the same size as p has), and alpha, beta, rho and rhon are scalars. All variables are defined as double precision.

This code runs at about 25% of the peak of the CM-5. Its performance can be improved by a few percents by replacing the statement with the CSHIFT's by a call to a routine from the CMSSL library. With this Poisson solver the overall speed of the 2D DNS code lies a little over 500 Mflop/s on a 16-node CM-5.

The CG-algorithm has two synchronization points, namely the two innerproducts in the computation of alpha and beta. There are various approaches reported to reduce the costs of these two innerproducts. In [7], for instance, it is proposed to postpone the update of p one iteration. Then, this update can be overlapped with the computation of alpha, and thus the iteration has one synchronization point less. The resulting method has the same numerical stability as the standard CG. We have implemented the CG-method with postponed update of p on the CM-5, and found that it is not faster than the standard implementation: the data parallel compiler does not recognize the possibility to overlap one innerproduct with an array update.

The shift to flop ratios for the CG-algorithm are 0.21 and 0.26 in two and three dimensions respectively. It is often remarked that the load plus store to flop ratio of CG is not very good. A closer look at the code generated by the data parallel CM-Fortran compiler shows that this ratio is not at all that bad: it is equal to 1.0 in 2D and equal to 0.9 in 3D.

As already remarked, the speed of convergence of the CG iteration can be improved by introducing an appropriate preconditioner. Here, an incomplete Choleski factorization is used as a preconditioner. This preconditioner introduces a recursion in both directions over the grid. A typical recursive relation is of the form:

```
x(i,j) = r(i,j) - a(i,j)*x(i-1,j) - b(i,j)*x(i,j-1).
```

See also Fig. 4.

The element x(i,j) depends on its previously computed neighbors in i and j direction. However, the elements x(i,j) on a diagonal i+j = d = constant depend only on values of x corresponding to a previous diagonal, and thus, in a diagonalwise ordering, the unknowns can be computed in parallel.

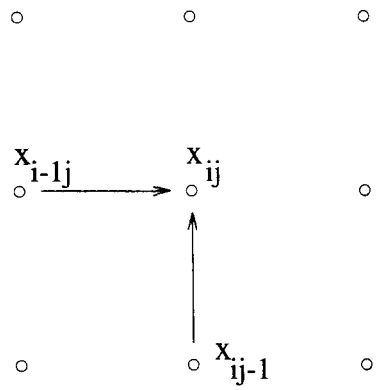

Fig. 4. The data flow introduced by the incomplete Choleski preconditioner.

This observation is explored on vector computers: the vectors correspond to diagonals of the grid. See e.g. [8]. On a parallel computer with local memory each processor can compute a part of a diagonal, if the unknowns are ordered, explicitly, in a diagonalwise manner. Suppose that x is a square array of N^2 elements. To store the elements of x diagonalwise we define an array xd of N*2N elements. The first diagonal (corresponding to d=1) of x is stored in the first column of xd, the second diagonal of x (d=2) is stored in the second column of xd, and so on.

When all arrays are stored in this manner, the data parallel code for the recursive relation reads

```
xd( ;1) = rd( ;1)
DO d = 2, 2*N
xd( ;d) = rd( ;d) - ad( ;d)* xd( ;d-1) - bd( ;d)
*CSHIFT(xd(;d-1), SHIFT=-1))
ENDDO
```

Here, all diagonals are stored in a 2D array. A slightly faster code can be obtained by storing each diagonal in a 1D array.

The diagonal re-ordering has several drawbacks. Owing to the variations of the lengths of the diagonals, some processors perform superfluous computations. Moreover, the 64 vector units of the 16-node CM-5 always have to be working on chunks of 8 elements. Hence, if N=512 then all 64 vector units can work on vectors of length 8. For much smaller N one cannot expect a high Mflop rate for the computation of the diagonals, since many of the vector units cannot do anything useful then.

The most serious drawback of the above approach is formed by the communication costs. The data flow for a diagonal update is sketched in Fig. 5.

To update the elements of the diagonal stored in the dth column of xd, a CSHIFT of the previous diagonal d-1 has to be performed. The number of data elements to be moved for this shift is extremely low. In an optimal implementation, only 15 elements are to be send from a processing node to its nearest neighbor to perform this shift. Consequently, the latency, i.e., the time for setting up the communication, dominates the communication time. In practice, the communication costs are excessively high: the CSHIFT in the DO-loop that computes xd(;d) takes almost all the time. This DO-loop causes the preconditioned CG method to run at only 1% of the peak of the CM-5. Consequently, this preconditioned CG method cannot compete with the unpreconditioned CG method. It may be

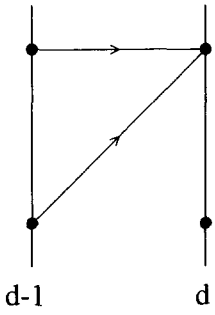

Fig. 5. The data flow introduced by the incomplete Choleski preconditioner, when the unknowns are diagonalwise re-ordered.

remarked that the long time required by the CSHIFT is partly due to a nonoptimal implementation of this operation. Yet, even if the CSHIFT would be implemented optimally fast, the estimated time (based on the hardware of the CM-5) for communication is too large for the diagonally re-ordered ICCG method to be competitive to the unpreconditioned one. Therefore, new preconditioners that are both numerically efficient and run at about 20% of peak on the CM-5 are to be developed.

6. Concluding remarks

- The data parallel programming style works well.
- Communication costs are higher for simulations in three spatial dimensions than in 2D.
- Our DNS code runs on a 16-node Connection Machine CM-5 at 23% (in 2D) and 13 % (in 3D) of the peak when the pressure is solved using the Conjugate Gradient method without preconditioning.
- An incomplete Choleski decomposition is not an efficient preconditioner on the CM-5. Preconditioners that are both numerically efficient and run at about 20% of peak are to be developed.

References

[1] W.C. Reynolds, The potential and limitations of direct and large eddy simulations, in: J.L. Lumley, ed., *Whither Turbulence? Turbulence at the Crossroads* (Springer, Berlin, 1990) 313–343.

[2] V.L. Peterson, J. Kim, T. Holst, G.S. Deiwert, D.M. Cooper, A.B. Watson and F.R. Bailey, Supercomputer requirements for selected disciplines important to aerospace, *J. IEEE* 77 (1989) 1038.

[3] R.W.C.P. Verstappen and A.E.P. Veldman, Direct numerical simulation of a 3D turbulent flow in a driven cavity, in: Wagner et al., eds., *Computational Fluid Dynamics '94* (Wiley, Chichester, 1994) 558–565.

[4] F.H. Harlow and J.E. Welsh, Numerical calculation of time-dependent viscous incompressible flow with free surface, *Phys. Fluids* 8 (1965) 2182–2189.

[5] I. Gustafson, A class of first-order factorization methods, *BIT* 18 (1978) 142–156.

[6] S. Eisenstat, Efficient implementation of a class of preconditioned Conjugate Gradient methods, *SIAM J. Sci. Statist. Comput.* 2 (1981) 1–4.

[7] J.W. Demmel, M.T. Heath and H.A. Van der Vorst, Parallel numerical linear algebra, *Acta Numerica* 2 (1993) 111–199.

[8] J.J. Dongarra, I.S. Duff, D.C. Sorensen and H.A. Van der Vorst, *Linear System Solving on Vector and Shared Memory Computers* (SIAM, Philadelphia, PA, 1991).